.

Studienarbeit

Entwicklung und Implementierung eines dynamischen Parsers für Ausdrücke

Christine Roehl
croehl@informatik.uni-rostock.de
Matrikelnummer: 091217673

Betreuer: Dipl.-Inf. Jürgen Schlegelmilch

Inhaltsverzeichnis

1 Einleitung

Das Objektbanksystem OSCAR ist ein Projekt, das 1988 an der Technischen Universität Clausthal gestartet wurde und unter der Leitung von Prof. Dr. Andreas Heuer seit 1994 an der Universität Rostock weitergeführt wird.

OSCAR basiert auf dem Datenmodell EXTREM, das neben Objektidentitäten und Klassenhierarchien auch eine Menge von Typkonstruktoren zur Beschreibung von Objektzustandstypen bereitstellt. Zum EXTREM-Modell wurden Anfragesprachen entwickelt, die das Modell komplett unterstützen. Dem Benutzer stehen neben der Objektalgebra ABRAXAS die SQL-artige Anfragesprache O^2QL und der Objektkalkül LILA zur Verfügung.

Im Moment hat OSCAR eine fest vorgegebene Menge von Datentypen und Operatoren, erste Ansätze für die Integration von Abstrakten Datentypen (ADTs) sind jedoch schon vorhanden. Ein Teilproblem dabei stellt das Parsen von Ausdrücken dar, z.B. in der Anfragesprache ABRAXAS. Zum Zeitpunkt der Compilierung des ABRAXAS-Parsers stehen weder die Menge der ADTs noch die ihrer Operatoren fest.

Im Rahmen dieser Studienarbeit entstand aufbauend auf dem Werkzeug Bison das Programm Jacob, das in der Lage ist, Ausdrücke zu parsen, deren Syntax ihm erst zur Laufzeit zur Verfügung steht. Das Programm zerfällt im wesentlichen in zwei Teile – die Parsergenerierungsfunktion und die Parsefunktion. Die Parsergenerierungsfunktion `jacob()` wertet die Syntaxregeln aus, die ihr in einer Notation ähnlich der in Bisongrammatik-Dateien verwendeten mitgeteilt wird.

2 Thema und Aufgabenstellung

- **Thema**: Implementierung eines dynamischen Parsers für Ausdrücke
- **Projekt**: OSCAR
- **Aufgabenstellung**: Die üblichen Parsergeneratoren wie `lex` und `yacc` können für die Erstellung eines dynamischen Ausdrucksparsers nicht verwendet werden, da sie Automaten konstruieren, deren Zustandsübergangsfunktionen zum Zeitpunkt der Compilierung feststehen müssen.

  Ziel dieser Arbeit ist es also, einen Kellerautomaten zu implementieren, bei dem diese Funktion erst zur Laufzeit durch eine Menge von Produktionsregeln aufgebaut wird.

  Die grobe Vorgehensweise des Parsens zerfällt in zwei Phasen:

  1. Die Parsetabelle wird aufgebaut, indem jeweils eine neue Produktionsregel (Syntaxdefinition für einen Operator) integriert wird. Dabei müssen Konflikte erkannt und gemeldet werden. Alle notwendigen Informationen wie Syntax, Vorrang oder Assoziativität müssen in der Syntaxdefinition eines Operators angegeben werden.
  2. Ein Ausdruck wird geparst, und dabei wird ein entsprechender Operatorbaum aufgebaut, der in den Knoten Verweise auf Operatoren und in den Blättern Verweise auf Operanden (Konstanten, Variablen oder Funktionsaufrufe) enthält.

  In weiteren Schritten (die nicht mehr Teil dieser Arbeit sind) kann dieser Baum dann von übergeordneten Parsern in ihre Operatorbäume integriert werden. Die Semantikprüfung des übergeordneten Parsers muß dann noch die Existenz der Operanden und die korrekte Typisierung des Operatorbaumes sicherstellen, bevor er optimiert und ausgewertet werden kann.

  Gegeben ist also eine Menge von Operatordefinitionen. Jede dieser Definitionen umfaßt folgende Elemente:

  1. eine Syntaxbeschreibung in Form einer Zeichenkette entsprechend dem Nichtterminalsymbol Operator:

     ```
     Klammer   ::= {|}|[|]
     Ziffer    ::= [0-9]
     Buchstabe ::= [a-z_A-Z]
     Parameter ::= #Ziffer
     Opsymbol  ::= -|!|$|&|/|=|?|^|@|+|*|~|<|>|||:
     Esc       ::= \
     Symbol    ::= Opsymbol|#|Esc
     Terminal  ::= Opsymbol|Esc Symbol
     Bezeichner ::= Buchstabe+
     Operator  ::= (Terminal|Parameter|Klammer|Bezeichner)+
     ```

Parameter bezeichnet einen Parameter des zu definierenden Operators. Die Nummer gibt die Position in der Parameterliste der dem Operator zugeordneten Funktion an. Die syntaktische Einschränkung auf maximal 10 Parameter ist schon aus Gründen der Lesbarkeit der möglichen Ausdrücke vernünftig und akzeptabel. Die Menge der verwendeten Nummern muß mit $n \in \mathbb{N}$ auch $(n-1)$ enthalten, außerdem darf jede Zahl nur einmal auftreten. Operatoren ohne Parameter sind nicht zulässig.

Um die Menge der möglichen Operatoren nicht unnötig einzuschränken, dient das Zeichen Esc zur Maskierung: Es macht aus dem folgenden Zeichen immer ein Terminalsymbol.

Als Operanden stehen – neben bereits erkannten Unterausdrücken – Konstanten, Variable und Funktionsaufrufe zur Verfügung. Ihre Syntax ist mit den gleichnamigen Nichtterminalsymbolen gegeben. Dabei ist zu beachten, daß der Parser alle Bezeichner als Variable betrachten soll, die nicht als Operator definiert sind. Man beachte, daß einige Zeichen reserviert sind. Neben den Ziffern und Buchstaben sind das die runden Klammern `()` für die Begrenzung von Unterausdrücken, das Komma `,` für die Konstruktion von Listen, der Doppelpunkt `:` für die Konstruktion von Marken, und das Semikolon `;` zur Markierung des Ausdruckendes.

2. Bei Operatoren mit zwei und mehr Parametern Angaben zu Assoziativität und Vorrang. Bei unären Operatoren stellt sich das Problem nicht.
3. Eine zugeordnete Funktion, gegeben durch ihren Namen und die Anzahl ihrer Parameter. Diese Anzahl muß gleich der Parameteranzahl in der Syntaxdefinition sein.

Die vom Parser zu erkennenden Ausdrücke haben dann die Form:

```
Konstante  ::=Ziffer+(.Ziffer+)?(e[+-]Ziffer+)?|
               "([^"]|\")*"|
               '[^']'
Variable   ::= Bezeichner
optMarke   ::= Bezeichner:|ε
Ausdruck   ::= (optMarke Ausdruck (,optMarke Ausdruck)*)|
               Konstante|Variable|(Ausdruck)|opAusdruck
```

Die Syntaxregeln für Ausdrücke, die Operatoren enthalten (Symbol opAusdruck), müssen aus den Operatordefinitionen gewonnen werden.

Mit dem Sequenzoperator `,` kann man Listen von Werten erzeugen, die durch Typ-Konvertierung in Tupel umgewandelt werden können. Syntaktisch tritt der Operator als linksassoziativer, binärer Operator in Infix-Notation auf, mit niedrigerer Priorität als alle anderen Operatoren.

Mit den Marken können die Elemente der durch den Sequenzoperator aufgebauten Listen benannt werden. Da im Datenmodell EXTREM nur die Namen der Komponenten eines Tupels, nicht aber ihre Reihenfolge eine Rolle spielen, ist es damit

möglich, die Argumente einer Funktion in beliebiger Reihenfolge anzugeben. Außerdem sehen Funktionsaufrufe auf Wunsch Smalltalk-ähnlicher aus. Im Falle unbenannter Elemente wird wieder die Reihenfolge für die Zuordnung verwendet.

Funktionsaufrufe sind Anwendungen eines unären Präfix-Operators in Form eines Bezeichners auf das Ergebnis eines Unterausdrucks. Man braucht also keine besondere Syntaxregel für sie, außerdem sind die Klammern dann nicht obligatorisch, d.h. man kann `sin x` schreiben, statt `sin(x)`.

Tupelstrukturierte Datentypen werden noch nicht unterstützt. Dazu ist eine Erweiterung des Nichtterminals Bezeichner notwendig, und zwar um Qualifizierer (qBezeichner ::= (Bezeichner.)*Bezeichner).

3 Grundlagen

Die Syntax einer Sprache gibt an, wie die Wörter der Sprache aufgebaut werden dürfen, und die Semantik definiert die Bedeutung der sprachlichen Konstrukte. Zur Beschreibung der Syntax einer Sprache benutzt man häufig Grammatiken. Hierbei sind die kontextfreien Grammatik von besonderer Bedeutung.

Ein **Alphabet** ist eine nichtleere Menge von Zeichen. Als **Wörter** über diesem Alphabet bezeichnet man beliebige Folgen von Zeichen aus dieser Menge, einschließlich des leeren Wortes ε. Eine **Sprache** über einem Alphabet ist eine Teilmenge der Menge M^* aller möglichen Wörter.

Eine **Grammatik** G ist ein 4-Tupel (N, T, P, S), wobei N und T Alphabete sind. Die Schnittmenge von N und T ist leer. N wird als Menge der **Nichtterminale** bezeichnet und T als Menge der **Terminale.** Im Folgenden werden wir in Übereinstimmung mit der beim Werkzeug Bison gebräuchlichen Terminologie Terminale auch als **Tokentypen** bezeichnen. Die Vereinigung beider Mengen V wollen wir Menge der **Symbole** nennen. Ein ausgezeichnetes Nichtterminal ist das **Startsymbol** S. P ist die Menge der **Produktionen** oder Regeln, wobei eine Regel ein Paar ist, dessen Elemente Wörter über der Menge der Symbole sind. Für $(\alpha, \beta) \in P$ schreiben wir auch $\alpha \to \beta$. α nennt man dann die linke Seite einer Regel und β die rechte.

Wenn bei allen Regeln einer Grammatik auf der linken Seite nur ein Nichtterminal steht und die rechte Seite von einem beliebigen Wort über der Menge der Symbole gebildet wird, nennt man die Grammatik (nach Chomsky) **kontextfrei**.

Die Anwendung einer Regel $\alpha \to \beta$ auf das Wort $w \in V$ besteht darin, daß in w das Teilwort α (an beliebiger Stelle) durch β ersetzt wird; Bezeichnung: $w \Rightarrow w'$. w' ist **ableitbar** aus w (Bezeichnung $w \overset{*}{\Rightarrow} w'$) genau dann, wenn $w = w'$ oder wenn eine endliche Folge von Wörtern über V $w_1, w_2, \ldots, w_n$ exisitiert mit $w_1 \Rightarrow w_2,\ w_2 \Rightarrow w_3,\ \ldots,\ w_{n-1} \Rightarrow w_n$ und $w = w_1 \wedge w_n = w'$.

Die von der Grammatik $G = (N, T, P, S)$ **erzeugte Sprache** $L(G)$ ist die Menge der Wörter v über V, für die es eine Ableitung $S \overset{*}{\Rightarrow} v$ gibt.

Eine Ableitung $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots w_n$ heißt **Rechtsableitung,** wenn bei jedem Ableitungsschritt das am weitesten rechts stehende Nichtterminal vermöge einer Regel aus P ersetzt wird. Es ist aber möglich, daß es für ein Wort $w \in L(G)$ verschiedene Rechtsableitungen gibt. Ist das der Fall, nennt man diese Grammatik G **mehrdeutig**.

Ein **Parsebaum** stellt graphisch dar, wie aus dem Startsymbol einer Grammatik ein Wort der Sprache abgeleitet wird. Der Wurzelknoten eines Parsebaumes repräsentiert das Startsymbol; in den Blättern befinden sich die Terminale, aus denen das Wort besteht. Alle Knoten des Baumes repräsentieren Nichtterminale. Wenn ein Knoten A von links nach rechts die Nachfolger $X_1, X_2, \ldots, X_n$ hat, so muß $A \to X_1 X_2 \ldots X_n$ eine Regel der Grammatik sein. Die Zeichenkette, die von links nach rechts alle Blätter des Parsebaumes enthält, bezeichnet man als **Ertrag** oder **Front** eines Baumes.

Ein **Parser** ist nun eine Funktion, die jedem String eine Menge von Parsebäumen zuordnet. Der Parser **akzeptiert** ein Wort, wenn er dem Wort mindestens einen Parsebaum zuordnet.

Für jede durch eine kontextfreie Grammatik erzeugbare Sprache existiert ein Kellerautomat, der diese Sprache akzeptiert.

Ein **Kellerautomat** ist ein 7-Tupel $(X, Z, \Gamma, z_A, K_A, f, Z_E)$, mit endlichen, nichtleeren Mengen X (Menge der Eingabezeichen), Z (Menge der Zustände) und Γ (Menge der Kellerzeichen). $z_A \in Z$ ist der Anfangszustand, $Z_E \subseteq Z$ ist die Menge der Endzustände, $K_A \in \Gamma$ das Anfangskellerzeichen und f eine partielle Funktion, $f : (X \cup \{\varepsilon\}) \times Z \times \Gamma \to Z \times \Gamma^*$. f erfüllt die folgende Bedingung: wenn $f(\varepsilon, z, k)$ definiert ist, so ist $\forall x \in X$ $f(x, z, k)$ nicht definiert.

Einen Kellerautomaten kann man sich vorstellen als abstrakte Maschine, die besteht aus einem Eingabeband, auf dem das Eingabewort steht und das nur vorwärts bewegt werden kann, einem Speicherband (auch Keller genannt), das hoch und runter bewegt werden kann und einer Steuereinheit mit Lesevorrichtung für das Eingabeband und mit Lese-Schreib-Vorrichtung für das Speicherband. Eingabe- und Speicherband bestehen aus Feldern, die jeweils maximal ein Eingabe- bzw. Kellerzeichen enthalten können. Die Lesevorrichtung für das Eingabeband ermöglicht das Lesen von jeweils einem Zeichen. Die Lese-Schreib-Vorrichtung für das Speicherband gestattet das Lesen eines Zeichens und das Schreiben eines Wortes aus Γ^* ab dem Feld (einschließlich), über dem der Schreib-Lese-Kopf steht (nach oben). Ein Zeichen auf dem Speicherband kann gelöscht werden, indem in das entsprechende Feld das leere Wort ε geschrieben wird. Die Steuereinheit bestimmt das Verhalten des Kellerautomaten, wobei sie sich der partiellen Übergangsfunktion f bedient.

Die **Konfiguration eines Kellerautomaten** beschreibt seinen Gesamtzustand durch ein Tripel $(w, z, \alpha), w \in X^*, z \in Z, \alpha \in \Gamma^*$. Eine Konfiguration kann wie folgt interpretiert werden: w ist das noch nicht gelesene Endstück des Eingabewortes auf dem Eingabeband. z ist der Zustand der Steuereinheit und α ist der Inhalt des Speicherbandes. Vermöge f kann nun eine Übergangsrelation $\vdash$ zwischen Konfigurationen definiert werden: $(w, z, K\alpha) \vdash (w, z', \beta\alpha)$, wenn $f(\varepsilon, z, K) = (z', \beta)$ ist und $(xw, z, K\alpha) \vdash (w, z', \beta\alpha)$, wenn $f(x, z, K) = (z', \beta)$ ist (wobei $x \in X$, $K \in \Gamma$ und $\alpha, \beta \in \Gamma^*$).

Die **Arbeitsweise eines Kellerautomaten** kann dann wie folgt bechrieben werden: Wenn das Speicherband leer ist, bleibt der Kellerautomat stehen; es sind keine Übergänge möglich. Wenn sich der Keller in Zustand z befindet, das obere Zeichen auf dem Speicherband K ist und $f(\varepsilon, z, K)$ definiert und gleich (z', β) ist, so geht der Kellerautomat in den Zustand z' über und K wird ersetzt durch β; das Eingabeband bleibt stehen, d.h. kein Zeichen wird gelesen. Wenn sich der Kellerautomat in der Konfiguration $(xw, z, K\alpha)$ befindet und $f(x, z, K) = (z', \beta)$, so geht der Kellerautomat in den Zustand z' über. K wird durch β ersetzt und das Eingabeband bewegt sich um ein Feld nach links, d.h. der Lesekopf um ein Feld nach rechts. Wenn keiner der Fälle zutrifft, bleibt der Kellerautomat stehen; kein Übergang ist möglich.

Sei $\overset{*}{\vdash}$ die reflexive transitive Hülle von $\vdash$. Dann wird $w \in X^*$ vom Kellerautomaten **akzeptiert,** wenn gilt $(w, z_A, K_A) \overset{*}{\vdash} (\varepsilon, z_E, \alpha)$ und $z_E \in Z_E, \alpha \in \Gamma^*$.

Bei der **Bottom-Up-Syntaxanalyse** wird versucht, einen Parsebaum, der bei den Blättern beginnend aufgebaut wird, für das Eingabewort zu erzeugen. Eine Methode dafür ist die **Shift-Reduce-Syntaxanalyse**. Dabei wird versucht, das Wort w auf das Startsymbol S zu reduzieren. In jedem Schritt werden Symbole, die mit einer rechten Seite

einer Regel übereinstimmen, durch die linke Seite der Regel ersetzt. Ein Problem dabei ist, zu erkennen, welche Symbole ersetzt werden müssen, um zum Startsymbol zu gelangen, und welche Regel angewandet werden soll, wenn es Regeln gibt, deren rechte Seiten übereinstimmen. Wenn in jedem Schritt die richtige Reduktion gewählt wurde, erzeugt man eine Rechtsableitung in umgekehrter Reihenfolge.

Ein Parser, d.h. ein Kellerautomat, der nach dieser Methode arbeitet, kann das nächste Eingabesymbol in den Keller (auf das Speicherband) **schieben,** die rechte Seite einer Regel, die oben auf dem Keller liegt, durch die linke Seite ersetzen – **reduzieren** –, das erfolgreiche Ende des Parsens melden (wenn das Eingabewort zum Startsymbol reduziert wurde) und Fehler erkennen. Diese Methode funktioniert nur für LR-Grammatiken (L bedeutet, die Eingabe wird von links nach rechts abgearbeitet, R bedeutet, eine Rechtsableitung wird gebildet). Das sind solche, wo der Parser immer entscheiden kann, ob er schieben oder reduzieren soll (keine schiebe/reduziere-Konflikte) und mit welcher Regel zu reduzieren ist (keine reduziere/reduziere-Koflikte). Aber LR-Grammatiken genügen, um praktisch alle Programmiersprachenkonstrukte zu beschreiben. (Literatur: [ASU88], [KV84])

4 Lösungsansätze

Ziel dieser Arbeit ist es also, einen Parser für Ausdrücke zu erstellen, die benutzerdefinierte Operatoren enthalten können, deren Syntax erst zur Laufzeit bekannt ist. Dazu wurden drei verschiedene Lösungsansätze untersucht.

Die erste Möglichkeit bei der Erstellung eines solchen dynamischen Parsers besteht in der Verwendung von Teilen der Sprache Transframe, die bereits Konzepte zur Definition neuer Operatoren durch den Nutzer enthält.

Der zweite Ansatz sieht vor, einen dynamischen Parser komplett zu entwerfen und zu implementieren. Ein Teilproblem dabei ist das Auflösen von Mehrdeutigkeiten. Dafür könnten die sogenannten Parsebaumfilter verwendet werden.

Bison zu erweitern, ein Werkzeug zum Schreiben von Compilern für Sprachen, deren Syntax allerdings schon zur Übersetzungszeit feststehen muß, ist der dritte untersuchte Ansatz.

In den folgenden Abschnitten werden diese drei Ansätze kurz vorgestellt, um dann anschließend begründen zu können, warum der dritte Ansatz gewählt wurde.

4.1 Transframe

Transframe ist eine C++-ähnliche Sprache von David L. Shang, die mittlerweile (leider) kommerziell vertrieben wird. Einem White-Paper von Shang ([Sha97])zufolge enthält sie neben vielen anderen Konzepten auch ein eigenes für Operatoren.

Folgende Möglichkeiten sind in Transframe für Operatoren vorgesehen:

- flexibles Format

  Das Format eines Operators ist nicht fix. Ein binärer Operator beispielsweise kann später auch als unärer Operator genutzt werden.

- unbegrenzte Art von Formaten

  Das Format eines Operators wird in der Interfacespezifikation durch den Benutzer völlig beliebig definiert.

- benutzerdefinierte Operatoren

  Es gibt eine Reihe von built-in-Operatoren mit vorgegebener Priorität und Assoziativität. Darüber hinaus kann der Benutzer aber jederzeit neue Operatoren hinzufügen.

4.2 Parsebaumfilter

Da der Benutzer seine Operatoren im Bezug auf die Syntax völlig frei definieren kann, kann beim Erstellen des dynamischen Parsers nicht davon ausgegangen werden, daß eine eindeutige kontextfreie Grammatik vorliegt. Ein Konzept zum Auflösen von Mehrdeutigkeiten muß also bei diesem Ansatz unbedingt integriert werden.

Eine mehrdeutige kontextfreie Grammatik (kfG), wie wir sie für unsere Ausdrücke verwenden, läßt verschiedene Interpretationen für einige Sätze der Sprache zu. Aber: Die Sprachdefinition sollte jedem String eine eindeutige Interpretation zuordnen. D.h. eine Sprachdefinition auf Basis einer kfG muß die passende Interpretation auswählen. Diese Auswahl kann, wie von Paul Klint und Eelco Visser (Universität Amsterdam) ([KV84]) vorgeschlagen, über den Begriff **Parsebaumfilter** spezifiziert werden.

Klint und Visser gehen davon aus, daß beim Auftreten von Mehrdeutigkeiten während des Parsens zunächst alle möglichen Parsebäume erzeugt werden und nennen das Parseergebnis – eine kompakte Darstellung einer Menge von Parsebäumen – Parsewald. Diesen Parsewald wollen sie in einem späteren Schritt verkleinern, indem sie die sogenannten Parsebaumfilter anwenden.

Daraus ergibt sich (abweichend von der in der Aufgabenstellung vorgeschlagenen Verfahrensweise) folgendes Vorgehen:

1. Parsen der Operatordefinitionen
2. Erstes Erkennen von Mehrdeutigkeiten
3. Aufbau eines Kellerautomaten, der wahrscheinlich immer noch mehrdeutig sein wird
4. Parsen von Ausdrücken und Erzeugen eines Parsewaldes
5. Anwendung von Filtern zum Beschneiden des Parsewaldes

4.2.1 Was ist ein Filter? – Eine informelle Einführung

Ein **Parsewald** ist eine kompaktere Darstellung einer Menge von Parsebäumen durch:

- gemeinsames Benutzen von Teilbäumen
- Zusammenfassen von Bäumen mit gleichem Ertrag zu einem Knoten

Parsewälder sind beschreibbar durch **Kontexte**. Das sind Parsebäume mit genau einem Vorkommen eines Loches. Die **Instanz eines Kontextes** wird erzeugt, indem das Loch durch einen Baum ersetzt wird. Wenn wir in einer Menge von Kontexten alle Löcher durch den gleichen Baum ersetzen, erhalten wir die **Instanz einer Menge von Kontexten**. Das entspricht einer gemeinsamen Benutzung von Teilbäumen. Das Zusammenfassen von Bäumen in einem einzigen Knoten kann nun durch die Instanzierung eines Kontextes mit einer Mengen von Parsebäumen dargestellt werden.

Ein **Filter** ist eine Funktion, die Mengen von Parsebäumen auf Mengen von Parsebäumen abbildet, wobei gefordert wird, daß die Ergebnismenge eine Teilmenge der Ausgangsmenge ist, d.h. daß die Menge der Parsebäume durch Anwendung eines Filters verkleinert wird. Filter werden häufig negativ definiert (Welche Bäume sind falsch?). Beispiele für Filterdefinitionen finden sich in der Arbeit von Klint und Visser.

Es handelt sich hierbei also um eine sehr allgemeine Definition, die sehr viele Funktionen als Filter zuläßt.

Per Definition werden Filter nach dem Parsen angewendet und sind dadurch parserunabhängig. Trotzdem ist es aus Effizienzgründen besser, einen Filter so früh wie möglich anzuwenden, um den Aufbau unnötiger Parsebäume zu vermeiden.

4.3 Bison anpassen

Die dritte Möglichkeit, einen dynamischen Parser zu erzeugen, ist, Bison an unsere Anforderungen anzupassen. Große Teile können hierbei von Bison übernommen werden (Parsen der Grammatikdefinitionsdatei, Erzeugen des Kellerautomaten). Ein weiterer Vorteil ist die freie Verfügbarkeit von Sourcen und Dokumentationen. Dies ist der Weg, den wir im Verlauf dieser Arbeit weiter verfolgen werden und auf den wir deshalb an dieser Stelle nicht genauer eingehen wollen.

4.4 Fazit

Die Aussagen des White-Papers von Shang und somit die Eignung von Transframe für unsere Zwecke können im Moment nicht überprüft werden, da weder Demoversionen noch Hinweise auf Realisierungsmöglichkeiten oder Implementationsdetails frei verfügbar sind. Deshalb können wir diesen Ansatz nicht verfolgen.

Das Papier von Klint Visser, das nur die theoretischen Grundlagen für die Arbeit mit Parsebaumfiltern schaffen will, enthält über die Definitionen hinaus lediglich einige theoretische Fallstudien (u.a. für Prioritäten, eine eingeschränkt erweiterbare Sprache und Pattern-Matching). Eine mögliche Implementation dieser Idee müßte also zunächst einen dynamischen Parser realisieren, der in der Lage ist, mehrere Parsebäume, genauer Parsewälder, gleichzeitig zu erzeugen und zu verwalten.

Neben diesem Parser, der zur Laufzeit die Operatordefinitionen liest und daraus die Parsetabellen für den mehrdeutigen Kellerautomaten zusammenstellt, müßte auch ein Konzept zur Umsetzung der Filteridee erarbeitet und implementiert werden.

Wenngleich dieser Ansatz sehr interessant aussieht und eine neue Herangehensweise darstellt, kommen wir doch mit der in den folgenden Kapiteln genauer beschriebenen Anpassung des Werkzeugs Bison schneller, einfacher und effizienter zum Ziel.

Um einen Einblick in die Arbeit Bisons zu erhalten und die notwendigen Änderungen begründen zu können, erfolgt im nächsten Kapitel eine Einführung in die Vorgehensweise von Bison, zunächst als Überblick, anschließend im Detail. Kapitel 6 beschreibt davon ausgehend alle notwendigen Änderungen, die an Bison vorgenommen werden mußten. Nach einer Zusammenfassung in Kapitel 7 wird in Kapitel 8 gezeigt, welche Aufgaben (im Rahmen einer nächsten Studienarbeit) noch anstehen, um den dynamischen Parser weiterzuverwenden.

5 Vorgehensweise von Bison

5.1 Überblick

Bison ist ein Werkzeug, mit dem man Parser erzeugen kann, die in der Lage sind, Sprachen zu parsen, die durch eine kontextfreie Grammatik definiert sind. Dafür wird durch Bison ein Kellerautomat erzeugt, der wie in Kapitel 3 auf Seite 8 beschrieben arbeitet.

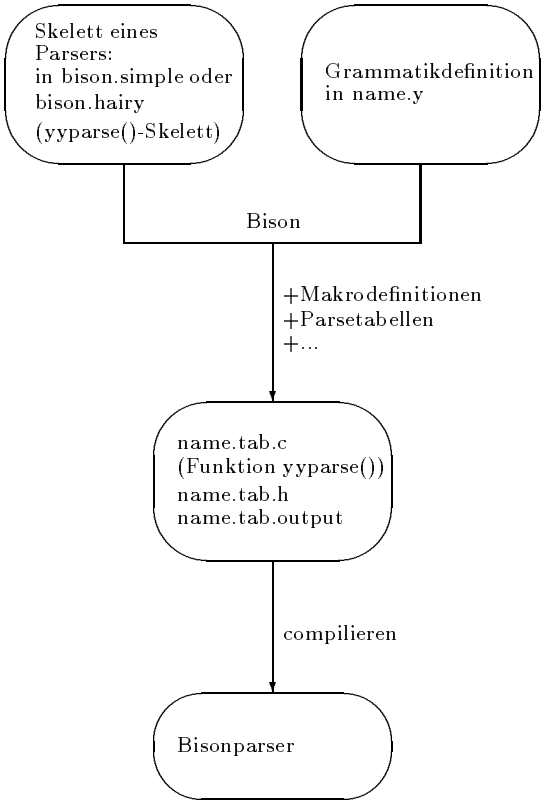

Abbildung 1: Bisons Vorgehensweise beim Erstellen eines Parsers

Der erste Schritt beim Erzeugen eines Parsers mit Bison ist das Erstellen einer Grammatikdefinitionsdatei (kurz: Grammatikdatei), die Bison als Eingabe benötigt. Üblicherweise ist diese Datei mit der Endung `.y` versehen.

Die Datei wird durch Bison eingelesen und ausgewertet. Die dabei gesammelten Informationen werden zusammen mit dem Gerüst der Funktion `yyparse()`, das in der Datei `bison.hairy` (für einen semantischen Parser) oder in `bison.simple` (sonst) zu finden ist, in Form von C-Code in eine Datei mit der Endung `.tab.c`. geschrieben. Diese von Bison als Ausgabe erzeugte Datei wird Bisonparser genannt.[1]

Auf diese Weise wird die Funktion `yyparse()` gebildet, die in der Lage ist, die durch die Grammatik beschriebene Sprache zu parsen, aber allein noch kein lauffähiges Programm darstellt.

Zusätzlich werden noch einige Funktionen benötigt, die durch den Nutzer bereitzustellen

[1] Das Werkzeug Bison und der Bisonparser sind also zwei verschiedene Dinge.

sind. Dazu gehört natürlich eine `main()`-Funktion, die den Parser aufrufen muß, weiterhin eine Fehlerreport-Funktion sowie eine Funktion, die das Scannen/Lexen des Eingabestroms übernimmt – `yylex()`. Wenn diese Funktion nicht in derselben Datei wie `yyparse()` steht, wird zudem eine Datei mit der Endung `.tab.h` benötigt, in der die Tokennummern definiert werden.[2]

Verdeutlicht wird diese Vorgehensweise in Abbildung 1.

5.2 Details

Welche Informationen werden nun von Bison während der Auswertung der Grammatikdefinitionsdatei gesammelt?

Ein zentrales Konzept von Bison sind „Aktionen“, die in der Grammatikdatei den einzelnen Regeln zugeordnet werden können. Sie bestehen aus C-Code und bieten dem Nutzer die Möglichkeit, beispielsweise den semantischen Wert des Nichtterminals auf der linken Seite der Regel zu berechnen. Die Aktionen werden von Bison zu einer `switch`-Anweisung zusammengefaßt und in die Bisonparserdatei geschrieben. Der Bisonparser führt dann später einen Aktionsrumpf immer dann aus, wenn er mit der zugehörigen Regel reduziert hat.

Außerdem werden Bison- und C-Deklarationen ausgewertet, und es wird entschieden, ob ein semantischer (vgl. Abschnitt 6.5.2 auf Seite 25) oder ein non-reentrant (vgl. Abschnitt 6.5.3 auf Seite 25) Parser geschrieben werden soll oder ob ein einfacher ausreicht. Die Tokendefinitionen werden eingelesen, der Typ von `yylval` – das ist die globale Variable, in der der semantische Wert des aktuellen Tokens gespeichert ist – wird festgelegt u.v.a.m.

Ferner werden die Parsetabellen erzeugt und in Form von C-Code als Vektoren in die Bisonparserdatei geschrieben.

[2] Die Angaben zu Endungen und das Präfix `yy` bzw. `YY` für Funktions- und Variablennamen bzw. für Makros beziehen sich auf die Voreinstellungen von Bison und können über Optionen verändert werden.

6 Notwendige Änderungen – Von Bison zu Jacob

Neben den Teilen, die unverändert von Bison übernommen werden können, müssen z.T. wesentliche Teile geändert werden. Ergebnis dieser Änderungen sind das Programm `jacob` und die Funktion `jacob()` (jetzt ein anderer Compiler-Compiler für Operatoren basierend auf Bison). Die `main()`-Funktion des Programms `jacob` tut nichts anderes als die Parsergenerierungsfunktion `jacob()` und anschließend die Parsefunktion `yyparse()` aufzurufen. So kann das Programm `jacob` auch nach erfolgter Einbindung der Funktion `jacob()` in das OSCAR-Projekt zum Testen von Operatordefinitionen verwendet werden.

Auf die notwendigen Änderungen auf dem Weg von Bison zu Jacob wollen wir in diesem Kapitel genauer eingehen.

6.1 Von der Operatordefinition zur Parsefunktion

Der Nutzer soll Jacob zur Laufzeit mitteilen, welche Operatoren er zulassen möchte, d.h. wie Ausdrücke gebildet werden können.

Die Syntax für Standardausdrücke (so wollen wir alle Ausdrücke bezeichnen, die keine nutzerdefinierten Operatoren enthalten, also z.B. Ausdrücke, die nur aus einem Bezeichner oder einer Konstanten bestehen) ist durch Regeln in der Datei `jacob.y` festgelegt.

Da diese Datei zur Laufzeit durch die Parsergenerierungsfunktion `jacob()` ausgewertet wird, bietet sich hier die erste Möglichkeit, Ausdrücke um neue Operatoren zu erweitern, indem über die Regeln für Standardausdrücke hinaus auch Regeln für Ausdrücke, die nutzerdefinierte Operatoren enthalten, aufgenommen werden. Wenn jedoch in dieser Datei syntaktische Fehler auftreten, hat das zur Folge, daß die Parsetabellen nicht korrekt oder nur unvollständig aufgebaut werden, was bedeutet, daß der später aufgerufene Parser (d.h. die Funktion `yyparse()`) sofort abbricht und nicht einmal in der Lage ist, Standardausdrücke zu parsen. Deswegen empfehlen wir die zweite Möglichkeit, dem Parser neue Operatoren bekannt zu machen.

Diese besteht darin, eine Datei mit Operatordefinitionen – `operatoren.def` – gemäß der in der Aufgabenstellung angegebenen Syntax erweitert um die Angabe von Funktionen (vgl. Abschnitt 6.3.2 auf Seite 21) und um die Angabe von Vorrangregeln (vgl. Abschnitt 6.5.1 auf Seite 24) zu erstellen.[3] Das im Rahmen dieser Arbeit erstellte Programm `op` erzeugt daraus die Datei `newrules.y` mit Grammatikregeln in Bisonsyntax für Ausdrücke, die aus diesen Operatoren gebildet werden können. Diese Datei wird von der Parsergenerierungsfunktion `jacob()` nach dem Lesen von `jacob.y` eingelesen. Die Aufrufsyntax von `op` sieht wie folgt aus: `op [-h] [-o output-file] input-file`.[4]

Das Programm `op` meldet jeden in der Operatordefinitionsdatei gefundenen Fehler und wandelt nur korrekte Definitionen in Grammatikregeln um, d.h. die Datei `newrules.y` enthält nur syntaktisch korrekte Grammatikregeln, wenn auch (beim Auftreten von Fehlern in einzelnen Operatordefinitionen) nicht unbedingt alle gewünschten. Wenn die Parsergenerierungsfunktion auf diese (eventuell unvollständige) Grammatikdatei angewen-

[3] Die vollständige Syntax ist im Kapitel 7 auf Seite 27 angegeben.

[4] D.h., daß natürlich auch andere Dateinamen als die hier vorgeschlagenen verwendet werden können.

det wird, ist der anschließend aufgerufene Parser zumindest in der Lage, einen Teil der gewünschten Ausdrücke zu parsen. Nur solche Ausdrücke, die Operatoren enthalten, deren Definition fehlerhaft war, werden Fehlermeldungen verursachen.

Da das Programm `op` unabhängig von Jacob zu benutzen ist, kann der Nutzer den Zyklus „Ändern der Operatordefinitionsdatei – Aufruf des Programms `op` (eventuell mit der Option `-o`)“ solange wiederholen, bis keine der Definitionen mehr einen Fehler enthält.

Die Grammatikregeln in der Datei `newrules.y` folgen weitestgehend der Syntax für Bisongrammatikregeln. So konnten das Einlesen und Parsen der Grammatikdateien `jacob.y` und `newrules.y` zum größten Teil von Bison unverändert übernommen werden.

Da jetzt aber zwei Dateien statt nur einer eingelesen werden müssen, mußte dafür gesorgt werden, daß nach dem Einlesen der Regeln aus `jacob.y` die Datei `newrules.y` gelesen wird. Deswegen werden anstelle der Funktionen `getc()` und `ungetc()` die Makros `GETC` und `UNGETC` verwendet. Diese Makros sorgen dafür, daß, wenn die globale Variable `use_modified_getc` gesetzt ist, `getc()` nicht mit `finput`, dem Filedeskriptor für `jacob.y`, sondern mit `fnewrules`, dem Filedeskriptor für `newrules.y`, als Parameter aufgerufen wird.

Außerdem sind in den Grammatikdateien jetzt auch die Angabe von `%left`, `%right` und `%nonassoc` innerhalb der Regelsektion (und nicht wie bisher nur im Bisondeklarationsteil, vgl. Abschnitt 6.5.1 auf Seite 24) sowie Funktionsvereinbarungen (vgl. Abschnitt 6.3.2 auf Seite 21) erlaubt. Die Einlese-Funktionen wurden dahingehend erweitert.

Das Scannen/Lexen der Ausdrücke muß nicht dynamisch realisiert werden, weil der Aufbau der syntaktischen Elemente schon zur Übersetzungszeit bekannt ist. Festgelegt wird er in der Datei `jacob.l`, aus der mit Hilfe von `flex` die Funktion `yylex()` gebildet wird.

6.2 C-Code-Generierung

Da Jacob die Grammatikregeln zur Laufzeit einlesen soll, können die Parsetabellen und alle anderen benötigten Funktionen nicht über den Umweg der Zwischendarstellung als C-Code, der erneut übersetzt werden muß, erzeugt werden.

Alles, was in die Datei mit dem Bisonparser, im folgenden auch Parserdatei oder Tabdatei genannt (Endung `.tab.c`), und in die Headerdatei, im folgenden auch Definitionsdatei genannt, weil sie hauptsächlich die Tokendefinitionen beinhaltet (Endung `.tab.h`), geschrieben wurde und anschließend compiliert werden mußte, muß nun zur Laufzeit, d.h. dynamisch, aufgebaut werden, ohne erneute Übersetzung.

Das betrifft insbesondere:

- C-Code in der Grammatikdatei

  Da die Grammatikdatei nun zur Laufzeit eingelesen werden soll, ist eine Übersetzung des C-Codes, der in Form von C-Deklarationen am Dateianfang (eingeschlossen in `%{` und `%}`), in Form von Aktionen oder in Form von zusätzlichem C-Code am Ende der Grammatikdefinitionsdatei vorkommen kann, nicht möglich.

  D.h. C-Code in der Grammatikdefinitionsdatei ist nicht mehr erlaubt, insbesondere

gibt es **keine Aktionen** mehr, und der Typ von `yylval` kann nicht mehr im C-Deklarationsteil angegeben werden![5]

Auf die Konsequenzen, die sich aus der Nichtverwendbarkeit der Aktionen ergeben, gehen wir im nächsten Abschnitt noch genauer ein.

- die Parsetabellen

  Das Füllen der Parsetabellen erfolgt bei Bison in einem Zwischenschritt. Die berechneten Werte können erst verwendet werden, wenn der erzeugte C-Code übersetzt und zum gewünschten Programm gelinkt wird.

  Anstatt die für die Parsetabellen berechneten Werte in Form eines Vektors als C-Code in eine Datei zu schreiben, werden die berechneten Werte nun sofort (zur Laufzeit) in einen entsprechenden Vektor geschrieben. Da die Tabellengröße nicht vorhersehbar ist (insbesondere, weil sie bei jedem Neulesen der Operatordefinitionen wachsen kann), muß der Speicherplatz für die Tabellen dynamisch verwaltet werden.

  Man kann auf die Vektoren, in denen die Werte der Parsetabellen stehen, nicht mehr direkt zugreifen, stattdessen ruft man entsprechende Funktionen auf. So ist z.B. statt `yytname[i]` jetzt `yytname(i)` zu verwenden.

- die Übersetzung in Abhängigkeit von Makros

  Bei der Auswertung der Kommandozeilenoptionen, die an Bison übergeben werden, sowie bei der Auswertung der Grammatikdatei werden globale Variablen gesetzt, z.B. die Variable `debugflag` auf den Wert 1, wenn die Option `-t` übergeben wurde (vgl. auch Abschnitt 6.4.1 auf Seite 22). Der Wert einiger Variablen wird in Form von Makrodefinitionen in die Bisonparserdatei geschrieben, z.B. `#define YYDEBUG 1`, wenn die Variable `debugflag` gesetzt ist.

  Precompileranweisungen in der Bisonparserdatei sorgen dafür, daß bestimmte Anweisungen nur übersetzt werden, wenn ein Makro gesetzt ist; z.B. wird der Code für Debug-Ausgaben nur übersetzt, wenn das Makro `YYDEBUG` einen Wert ungleich 0 hat.

  Für Jacob muß der gesamte C-Sourcecode übersetzt werden, da eine Codeveränderung zur Laufzeit nicht möglich ist. Die Ausführung der einzelnen Anweisungen jedoch muß von den Variablen abhängig sein, deren Werte vorher den Wert der entsprechenden Makros bestimmten. `#if YYDEBUG != 0` muß z.B. durch `if (debugflag)` ersetzt werden.[6]

- das `yyparse()`-Skelett,

[5] Der Typ von `yylval` wird nun in `template.h` festgelegt und kann dort geändert werden. Ebenso können die C-Deklarationen, die ursprünglich zwischen `%{` und `%}` standen, nun über `template.h` eingebunden werden. Allerdings wird dann natürlich eine erneute Übersetzung nötig.

[6] Die Ausgabe der Debugmeldungen ist bei Bison zusätzlich noch vom Flag `yydebug` abhängig, das der Benutzer vor dem Aufruf der Parsefunktion setzen kann. Nach der Ersetzung von `YYDEBUG` durch `debugflag` erhalten wir also: `if (debugflag) if (yydebug) fprintf(...) ...`. Weil es nicht sinvoll ist, zwei Variablen für einunddenselben Zweck zu verwenden, wurde `debugflag` durch `yydebug` ersetzt und eine `if`-Abfrage weggelassen.

das zusammen mit dem C-Code der Aktionen in die Bisonparserdatei kopiert wird, muß nun eine vollwertige Funktion werden, die nicht auf die statischen Tabellen, sondern auf die dynamischen zugreift. Das Parsegerüst wird nicht mehr aus einer der beiden Dateien `bison.simple` oder `bison.hairy` kopiert, sondern steht in Form einer „richtigen“ C-Funktion in der Datei `bison.c` zur Verfügung, die sich im gleichen Verzeichnis wie alle anderen C-Quelldateien befindet.

- die Definitionsdatei

  In diese Datei, die üblicherweise die Endung `.tab.h` hat, wird von Bison, wenn er mit der Option `-d` aufgerufen wird, für jedes Token eine `#define`-Anweisung geschrieben, z.B. `#define TK_BEZEICHNER 258`. Die so vereinbarten Tokennummer werden von `yylex()` zurückgegeben. Dazu muß diese Datei in die von `flex` erzeugte Datei, die die `yylex()`-Funktion enthält, über `#include` eingebunden werden. Jetzt, da wir nicht mehr das Programm Bison verwenden, sondern die Funktion `jacob()`, um den Parser zu erzeugen, ist dieser Weg nicht mehr möglich, weil die Funktion `yylex()`, die die Definitionsdatei einbindet, und die Funktion `jacob()`, die diese Datei erzeugt, zum gleichen Programmpaket gehören. Das Programm würde also zur Laufzeit eine zu seinem Sourcecode gehörende Datei verändern.

  Deswegen kann es diese Datei für Jacob nicht mehr geben. Die Ermittlung des Rückgabewertes in `yylex()` erfolgt aus diesem Grund über einen Umweg: Zunächst wird die von `jacob()` intern verwendete Tokennummer ermittelt. Diese entspricht dem Index des Tokennamen (z.B. `TK_BEZEICHNER`) im Vektor mit allen Tokennamen und kann mit Hilfe der Funktion `yytname()` ermittelt werden. Die interne Nummer wird mit der Funktion `yytoknum()` in die entsprechende `yylex()`-Tokennummer umgerechnet, die `jacob()` als Rückgabewert erwartet.

  Über die Definition der Tokennummern hinaus enthält die Datei die Definitionen für `YYSTYPE`, den Typ von `yylval` (semantischer Wert eines Tokens), und für `YYLTYPE`, den Typ der Elemente des Zeilennummernstacks, sowie einige `extern`-Deklarationen. Diese befinden sich nun in der Datei `jacob.h`.

6.3 Aktionen

Die zwei Aufgaben, für die wir normalerweise Aktionen verwenden würden, – die Erstellung eines Parsebaums und das Berechnen der semantischen Werte der Symbole – müssen, obwohl es keine Aktionen mehr gibt, erledigt werden. Wie – darauf wollen wir in den nächsten beiden Abschnitten eingehen.

6.3.1 Aufbau des Parsebaum

Um den Parsebaum aufzubauen, verwenden wir **eine** Aktion, die für alle Regeln gleich ist, die also bei jeder Reduktion ausgeführt wird.

Ein Baum besteht aus Knoten, die als Sonderfall Wurzel oder Blätter sein können. Repräsentiert wird der Baum durch einen Zeiger auf seine Wurzel.

Ein Knoten steht für genau ein Symbol (Terminal oder Nichtterminal). Er soll Informationen über das Element wie semantischen Wert und Name der auszuführenden Funktion (siehe nächsten Abschnitt) speichern und außerdem, wenn vorhanden, auf seine Nachfolgerknoten verweisen.

Verwaltet werden die Syntaxbäume der einzelnen Symbole über den Valuestack, der für nichts anderes mehr benötigt wird, weil es keine Aktionen und somit auch keine Referenzen `$1`, `$2`, ... mehr gibt, die dort ursprünglich verwaltet wurden. `yyval` (semantischer Wert der Nichtterminale, wird von den Aktionen geliefert) und `yylval` (semantischer Wert der Terminale, wird von `yylex()` geliefert) müssen dann vom gleichen Typ sein wie ein Knoten. Beim Reduzieren wird aus den Teilbäumen für die Symbole der rechten Seite, die auf dem Valuestack abgelegt sind, der Parsebaum für das Nichtterminal der linken Seite einer Regel erzeugt. Beim Schieben eines Tokens wird ein Blatt erzeugt und dieses auf dem Stack abgelegt.

Der Infoteil eines Knotens ist der semantische Wert des zugehörigen Symbols. Er wird zunächst in die Variable `yyinfo` geschrieben und von dort beim Schieben oder Reduzieren in den entsprechenden Knoten kopiert.

Die für den Aufbau und die Verwaltung des Parsebaumes benötigten Funktionen befindet sich in der neuerstellten Datei `tree.c`.

Der Typ eines Knotens ist in der Datei `jacob.h` wie folgt deklariert:

```
typedef struct node {
  YYSTYPE info;
  struct node **childs;
} node_t;
```

In der Datei `template.h`, die dazu dient Voreinstellungen vorzunehmen, wird der Typ des Infoteils eines Knotens deklariert:

```
typedef
  struct{
    union {
      double d;
      char s[500];
      funcdesc *func;
    } val;
    enum { is_double, is_string } type;
  } semantic_t;
#define YYSTYPE semantic_t
```

Die Elemente `d` und `s` der `union val` dienen zur Aufnahme des semantischen Wertes. Sie sollen durch die Funktion gesetzt werden, die beim Reduzieren mit einer Regel ausgeführt wird (siehe nächsten Abschnitt). Ein Zeiger auf die Funktion soll später in der Variable `func` stehen. Im Moment steht dort nur der Funktionsname und eine Liste mit den Werten ihrer Parameter. Der Typ `funcdesc` wird ebenfalls in der Datei `template.h` deklariert.

6.3.2 Verwendung von Funktionen

Die Berechnung des semantischen Wertes eines Nichtterminals soll statt der Aktion die zum Operator gehörende Funktion übernehmen. Diese Funktion, die in der Operatordefinitionsdatei – `operatoren.def` – angegeben wird, muß auch in der daraus erzeugten Grammatikdatei – `newrules.y` – erscheinen, um sie Jacob bekannt zu machen. Daraus ergibt sich, daß die Syntax einer Grammatikdatei um Funktionsvereinbarungen erweitert werden muß und zwar so, daß Funktionsvereinbarungen angegeben werden können, aber nicht müssen. Denn es ist sicher nicht sinnvoll, für die Regel `ausdruck : TK_KONSTANTE` die Angabe einer Funktion zu fordern.

Wenn die Angabe einer Funktion am Regelende fehlt, soll der semantische Wert des Nichtterminals auf der linken Seite (analog zu fehlenden Aktionen) vom Wert des ersten Nichtterminals auf der rechten Seite übernommen werden.

Die Syntax der Funktionsvereinbarung in der Operatordefinitionsdatei und in den Grammatikdateien stimmen überein: Das Schlüsselwort `function` wird gefolgt vom Funktionsnamen[7] und einer Parameterliste. Über diese Parameterliste kann innerhalb der Funktion auf die semantischen Werte der Symbole auf der rechten Seite der Regel zugegriffen werden.

Die Parameter können analog zu `$1`, `$2`, ... angegeben werden, die in den Aktionen verwendet wurden. Sie beginnen allerdings nicht mit `$`, sondern mit `#`, um Konsistenz zur Operatordefinitionsdatei zu bewahren. Ein `#2` in der Parameterliste bewirkt die Übergabe des semantischen Wertes des zweiten Ausdrucks auf der rechten Seite an die Funktion.

Um den Nutzer nicht unnötig einzuschränken, muß eine Funktion nicht zwingend die gleiche Anzahl von Parametern wie der zugehörige Operator haben; es dürfen nur nicht mehr sein. Auf diese Weise können auch Funktionen, die konstante Werte liefern, realisiert werden.

Außerdem ist es, abweichend von den in der Aufgabenstellung geforderten Bedingungen, nicht erforderlich, daß die Menge der verwendeten Nummern mit $n \in I\!N$ auch $(n - 1)$ enthalten muß. Bei Auswertung der Operatordefinitionsdatei durch das Programm `op` wird lediglich überprüft, ob die Parameter in der Liste auch in der Definition der Operatorensyntax vorkommen. Es ist trotzdem gewährleistet, daß in die Datei mit den neuen Grammatikregeln korrekte Parameternummern eingetragen werden.

Beim Einlesen der Grammatikregeln durch Jacob wird überprüft, ob sich die Parameternummern auf Ausdrücke und nicht auf andere syntaktische Elemente beziehen. Es kann hier aber nicht überprüft werden, ob sie sich auf den gewünschten Ausdruck beziehen.

Ein Beispiel soll das illustrieren: Wie schon in Abschnitt 6.1 auf Seite 16 angeführt, können neue Operatordefinitionen dem Parser auf zwei verschiedenen Wegen bekannt gemacht werden. Ein Nutzer wählt die Variante, die Datei `jacob.y` um Regeln für neue Operatoren zu erweitern und nimmt folgende Regel auf: `ausdruck : "myOp" ausdruck ausdruck`. Wenn er nun fälschlich annimmt, daß die Numerierung der Elemente der rechten Seite

[7] In der Grammatikdatei muß der Funktionsname in doppelten Anführungszeichen `"` eingeschlossen sein, damit beim Parsen einer Regel erkannt wird, daß es sich hierbei nicht um den Namen eines Tokens oder eines Nichtterminalsymbols handelt.

einer Regel mit 0 beginnt, verweist er mit `#2` auf den ersten Ausdruck (das zweite Elemenet auf der rechten Seite), obwohl er den zweiten meinte. Solche Verwechslungen sind ausgeschlossen, wenn stattdessen die Operatordefinition `define operator myOp #1 #2 function myFunc (#1,#2)` erstellt wird. Das Programm `op` erstellt daraus die korrekte Regel `ausdruck : "myOp" ausdruck ausdruck function "myFunc"(#2 , #3);`

Auch deswegen wird, wie schon in Abschnitt 6.1 auf Seite 16, empfohlen, in der Datei mit Regeln für Standardausdrücke keine Regeln für Operatoren aufzunehmen.

Die Funktion soll von der neuen (für alle Regeln gleichen) Aktion aus aufgerufen werden. Im Moment wird jedoch nur der Funktionsname ausgegeben, weil das dynamische Linken der Funktionen nicht mehr Bestandteil dieser Arbeit ist.

6.4 Funktion statt Programm

Wir verwenden für die Generierung des Parsers nicht länger ein Programm (Bison), sondern eine Funktion (`jacob()`), die bei Bedarf, d.h. wenn neue Operatordefinitionen eingelesen werden sollen, aufgerufen wird. Das hat drei Änderungen zur Folge, auf die wir in diesem Abschnitt eingehen wollen.

6.4.1 Optionen

An die Funktion können keine Optionen wie an ein Programm übergeben werden. Alle Änderungen von Voreinstellungen, die auf diesem Wege vorgenommen wurden, müssen nun in der Datei `template.h` erfolgen, d.h. die entsprechenden Makros sind dort direkt zu verändern, vgl. Tabelle 1.

Option	Langer Optionsname	Variable	Makroname
-t	--debug	yydebug	YYDEBUGFLAG
-b	--file-prefix	spec_file_prefix	SPEC_FILE_PREFIX
	--fixed-output-files	fixed_out_files	FIXED_OUTFILES
-o	--output	spec_outfile	SPEC_OUTFILE
-o	--output-file	spec_outfile	SPEC_OUTFILE
-r	--raw	rawtoknumflag	RAWTOKNUMFLAG
-v	--verbose	verboseflag	VERBOSEFLAG
-y	--yacc	fixed_out_files	FIXED_OUT_FILES

Tabelle 1: Verwendung von Variablen anstelle von Optionen

Einige der Optionen sind jedoch überflüssig geworden und können deshalb nun nicht auf diese Weise nachgebildet werden, vgl. Tabelle 2.

`defines` wird nicht weiter unterstützt, weil die Definitionsdatei jetzt nicht mehr erzeugt wird, vgl. Abschnitt 6.2 auf Seite 17.

`help` und `versions` sind für eine Funktion nicht sinnvoll und werden deswegen nicht mehr benötigt.

Option	Langer Optionsname	Veränderte Variable
-d	--defines	definesflag
-h	--help	
-n	--no-parser	no_parser_flag
-p	--name_prefix	spec_name_prefix
-l	--no-lines	no_lines_flag
-k	--token-table	toknumflag
-v	--version	

Tabelle 2: Nicht mehr zu verwendende Optionen und Variablen

`no-parser` wurde benutzt, wenn kein Parser erzeugt werden sollte. Die Parsetabellen wurden dann in die Tabdatei und die Aktionen in Form einer `switch`-Anweisung in eine Datei mit der Endung `.act` geschrieben. Diese Dateien, und somit diese Option, wollen wir nicht verwenden.

`spec-name-prefix` wurde verwendet, um ein neues Präfix für die Variablen und Funktionen, deren Namen mit yy begannen, anzugeben. Das wird dann notwendig, wenn mehr als ein Parser in einem Programm verwendet wird, um Konflikte zwischen den entsprechenden Variablen und Funktionen zu umgehen. Die Umbenennung kann natürlich nicht zur Laufzeit erfolgen und muß deswegen von Hand vorgenommen werden. Dazu sind für `yyparse, yylex, yyerror, yylval, yychar, yydebug, yynerrs` Makrodefinitionen in die Datei `template.h` zu schreiben, z.B. `#define yyparse xxparse`. Ein Konflikt mit allen anderen Variablen und Makros sollte laut Bisondokumentation nicht auftreten.

Wenn die Option `--no-lines` an Bison übergeben wurde, erzeugte er keine `#line`-Anweisungen in der Parserdatei, weil einige Compiler damit Probleme haben. Da wir diese Datei nun nicht mehr erzeugen wollen, wird diese Option bedeutungslos.

Die Option `--token-table` bewirkte die Ausgabe des Vektors `yytname` mit einer Liste aller Tokennamen, geordnet nach ihrer Bisontokennummer und des Vektors `yytoknum`, der für die entsprechenden Einträge in `yytname` die `yylex()`-Tokennummern enthält, sowie die Ausgabe von Definitionen für `YYNTOKENS`, `YYNNTS`, `YYNRULES` und `YYNSTATES`. Die Makros werden nicht mehr gebraucht, an ihrer Stelle können die Variablen verwendet werden, mit deren Werten sie belegt worden waren. Aber die beiden Tabellen werden jetzt immer, unabhängig von Flags, benötigt. Sie ermöglichen `yylex()` das Erkennen der Bezeichner, die als Bestandteil einer Operatordefinition eine besondere Bedeutung haben.[8]

Die Variable `infile` wurde ebenfalls durch das Auslesen der Kommandozeilenargumente gewonnen und muß jetzt bei Bedarf über das Makro INFILE in `template.h` verändert werden, ebenso wie die neue Variable `newrulesfile` (Makro NEWRULESFILE), die angibt, in welche Datei das Programm `op` die neuen Regeln geschrieben hat.

[8]Die Datei mit den neuen Regeln für Ausdrücke kann Bezeichner enthalten, z.B. `ausdruck : "sin" ausdruck function "sin"(#2)`. Beim Parsen des Ausdrucks `sin 3.14` muß `yylex()` davon abgehalten werden, `TK_BEZEICHNER` für `"sin"` zurückzugeben. Dazu muß `yylex()` mit Hilfe von `yytname()` die Bisontokennummer von `"sin"` ermitteln und diese anschließend mit Hilfe von `yytoknum()` in seine eigene Tokennummer umwandeln, denn diese erwartet der Parser.

6.4.2 Programmabbruch

In Bison wird beim Auftreten kritischer Fehler in beliebigen Verschachtelungstiefen sowie nach erfolgreicher Beendigung des Parsens die Funktion `done()` aufgerufen, die neben verschiedenen Aufräumarbeiten zur Aufgabe hat, das Programm mit `exit()` zu beenden. Dieses Verhalten ist nun nicht mehr erwünscht – das fehlerhafte oder erfolgreiche Parsen eines Ausdrucks darf nicht das gesamte Programm beenden! Deswegen kann `exit()` nicht länger verwendet werden.

Eine Möglichkeit, `exit()` zu umgehen, wäre gewesen, nach jedem Funktionsaufruf zu überprüfen, ob während der Funktionsausführung ein Fehler festgestellt wurde [9], um dann die aufrufende Funktion ebenfalls zu beenden. Dieser Weg scheint jedoch in Anbetracht der hohen Zahl der Funktionsaufrufe, d.h. der Verschachtelungstiefe, zu aufwendig und fehleranfällig. Deswegen wurde die Variante mit `setjmp/ longjmp` gewählt, dem C-Mechanismus für Sprünge über Block- und Dateigrenzen: `done()` sorgt jetzt dafür, daß nach dem (fehlerhaften oder erfolgreichen) Beenden des Parsens an das Ende der Parsergenerierungsfunktion `jacob()` gesprungen wird.

In `done()` wird außerdem die globale Variable `failure` gesetzt, wenn beim Einlesen der Grammatikdateien ein Fehler aufgetreten ist. Der Wert von `failure` wird von der Parsergenerierungsfunktion zurückgegeben.

6.4.3 Initialisierung globaler Variablen

Durch die Umwandlung Bisons in eine Funktion, die auch mehrmals nacheinander aufgerufen werden kann, gewinnt der Umgang mit globalen Variablen an Bedeutung. Eine fehlende Initialisierung, d.h. z.B. eine Initialisierung, die im Zuge einer Variablendeklaration erfolgte und nun infolge der Umwandlung in eine Funktion nicht mehr vor jedem Aufruf von `jacob()` durchgeführt wird, kann eine nicht korrekte Programmabarbeitung und im Extremfall einen Programmabsturz zur Folge haben.

Fehlende Initialisierungen, aber auch fehlende Speicherfreigaben, waren also im Zuge der Umwandlung des Programms Bison in die Funktion `jacob()` aufzuspüren und zu beheben.

6.5 Deklarationen

6.5.1 Assoziativität und Vorrang

Jacob soll auch Assoziativität und Vorrang benutzen, um Mehrdeutigkeiten aufzulösen. Wie aber kann man ihm diese mitteilen?

Zunächst müssen wir dazu Angaben in der Operatordefinitionsdatei machen. Das erfolgt analog zu den Assoziativitätsdeklarationen im Deklarationsteil einer Bisongrammatikdatei. Jede dieser Deklarationen für Links-, Rechts- oder Nichtassoziativität besteht aus einem Schlüsselwort (`%left`, `%right` oder `%nonassoc`) gefolgt von einer Tokenliste. Die

[9] Das könnte z.B. durch einen zusätzlichen Parameter angezeigt werden.

Token innerhalb einer Liste besitzen die gleiche Priorität. Wenn zwei Token in verschiedenen Assoziativitätsdeklarationen stehen, hat dasjenige Token die höhere Priorität, welches später deklariert wird.

Von der Operatordefinitionsdatei werden die Informationen in die Grammatikdatei mit den neuen Regeln geschrieben (`newrules.y`). Da das Lesen aus dieser Datei und aus der Datei mit den Standardausdrücken (`jacob.y`) aber so implementiert ist, als würde es sich dabei um eine einzige Datei handeln, stehen die Deklarationen nun nicht mehr im Deklarationsteil, sondern zwischen den Grammatikregeln, selbst wenn sie an den Anfang von `newrules.y` geschrieben wurden.

Um ein korrektes Parsen der Deklarationen auch hier zu ermöglichen, wurde die Funktion, die das Einlesen der Grammatikregeln übernimmt – `readgram()` – so verändert, daß am Regelanfang auch ein `%`, gefolgt von `left`, `right` oder `nonassoc`, akzeptiert wird.

Da die Funktion, die die Assoziativitäts-Deklarationen parst, davon ausgeht, daß nach einer Deklaration auf alle Fälle noch ein `%` folgt – um den Anfang der nächsten Deklaration oder das Ende des Deklarationsteils mit `%}` anzuzeigen – muß eine Deklaration, die statt im Deklarationsteil zwischen den Grammatikregeln steht, mit `%` abgeschlossen werden.

6.5.2 Einfacher vs. semantischer Parser

Bei Bison konnte der Nutzer sich entscheiden, ob er einen einfachen oder einen komplexeren Parser, den sogenannten semantischen Parser, erzeugen wollte. Letzterer bietet u.a. eine verbesserte Fehlerbehandlungsstrategie.

Für unsere Zwecke genügt der einfache Parser (in `bison.simple`). Deswegen erscheint eine Änderung des semantischen Parsers (in `bison.hairy`) nicht sinnvoll.[10] Alle Variablen und Funktionen, die nur benutzt wurden, wenn ein semantischer Parser erzeugt werden sollte, wurden deshalb entfernt. Dazu gehören u.a.:

- guards, die eine verbesserte Fehlerbehandlung ermöglichen
- die Tabelle `yystos[ ]`[11] und die Funktion `output_stos()`
- die Funktion `open_extra_files()`, die zusätzliche Dateien öffnet.

6.5.3 wiedereintrittsfest (reentrant) vs. nicht wiedereintrittsfest (non-reentrant)

Beim rekursiven Parsen oder in einer Multithread-Umgebung, d.h., wenn die Funktion `yyparse()` ein zweites Mal aufgerufen werden kann, während der erste Aufruf von `yyparse()` noch aktiv ist, wird ein wiedereintrittsfester Parser benötigt.

Bison erzeugt normalerweise keine wiedereintrittsfeste Funktion, weil z.B. die Variablen `yylval` und `yylloc`, die für die Kommunikation zwischen den Funktionen `yylex()` und `yyparse()` benötigt werden, statisch alloziert werden.

[10] Das würde den Programmieraufwand mehr als verdoppeln.

[11] `yystos[s]` gibt die Nummer des Symbols an, das zum Zustand s führt

Die Deklaration `%pure_parser` sorgt jedoch dafür, daß Bison einen wiedereintrittsfesten Parser erzeugt, d.h. einen, in dem keine statischen Variablen verwendet werden. Dadurch verändert sich u.a. die Schnittstelle zur Funktion `yylex()`.

Wenn `%pure_parser` gefunden wird, wird `#define YYPURE 1` in die Tabdatei geschrieben. In `bison.simple` bzw. in `bison.hairy` wird dann in Abhängigkeit von `YYPURE` unterschiedlicher Code erzeugt. Die Funktion `yylex()` hat z.B. Parameter, wenn `YYPURE` definiert ist, und keine Parameter, wenn `YYPURE` nicht definiert ist. Wenn `YYPURE` nicht definiert ist, werden die Variablen `yychar`, `yylval`, `yylloc` und `yynerrs` global deklariert. Im anderen Fall sind sie lokale Variablen.

Man kann also nicht wie bei einigen anderen Makros den Code so ändern, daß statt der Abhängigkeit von Makros eine Abhängigkeit von „echten“ Variablen entsteht. Das bedeutet, daß die Entscheidung, ob ein wiedereintrittsfester oder ein nicht wiedereintrittsfester Parser erzeugt werden soll, nicht erst zur Laufzeit erfolgen kann.

Ein nicht wiedereintrittsfester Parser genügt unseren Anforderungen. Außerdem ist er übersichtlicher und somit einfacher zu warten. Deswegen wurde mit der Funktion `jacob()` ein nicht wiedereintrittsfesten Parser implementiert.

Die Deklaration `%pure_parser` ist also nicht mehr verwendbar, und alle Variablen und Funktionen, die nur für den wiedereintrittsfesten Parser benötigt wurden, konnten entfernt werden.

6.6 Überflüssiger Code

Bison ist darauf ausgerichtet, auf möglichst vielen Plattformen zu laufen, weswegen der Sourcecode viele Anweisungen enthält, die beispielsweise für eine korrekte Arbeitsweise auf MS-DOS- oder VMS-Rechner benötigt werden. Da eine Unterstützung dieser Plattformen durch Jacob nicht notwendig ist, wurde der besseren Übersicht halber dieser Code entfernt, ebenso wie Code, der auskommentiert oder in `#if 0` und `#endif` geschachtelt war.

7 Zusammenfassung

Aus dem bisher Gesagten ergibt sich folgende schematische Vorgehensweise für das Erstellen einer Parsefunktion mit Hilfe von Jacob:

1. Erstellen einer Datei (`jacob.y`) mit Regeln für Standardausdrücke gemäß den Syntaxregeln für eine Bisongrammatikdatei, allerdings mit folgenden Einschränkungen:

   Die Bisondeklarationen `%guard`, `%semantic_parser`, `%pure_parser`, `%union`, `%no_lines`, `%token_table` können nicht mehr verwendet werden.

   Diese Datei darf keinen C-Code mehr enthalten, d.h. keine C-Deklarationen am Dateianfang, keinen Programmcode nach dem zweiten `%%` und keine Aktionen. Stattdessen darf am Regelende eine Funktion gemäß den im nächsten Punkt angegeben Regeln für das Nichtterminalsymbol Funktion angegeben werden, die ausgeführt werden soll, wenn mit dieser Regel reduziert wird. Dabei ist darauf zu achten, daß der Funktionsname, im Gegensatz zur Operatordefinitionsdatei, in " einzuschließen ist.

2. Erstellen einer Datei mit neuen Operatordefinitionen (`operatoren.def`). Die Syntaxregeln für Operatordefinitionen aus Kapitel 2 auf Seite 5 wurde um die Angabe einer Funktion und um Assoziativitätsdeklarationen erweitert (Startsymbol ist Zeile):

Klammer	::= {|}|[|]
Ziffer	::= [0-9]
Buchstabe	::= [a-z_A-Z]
Parameter	::= #Ziffer
Opsymbol	::= -|!|$|&|/|=|?|^|@|+|*|~|<|>|||:
Esc	::= \
Symbol	::= Opsymbol|#|Esc
Terminal	::= Opsymbol|Esc Symbol
Bezeichner	::= Buchstabe+
Operatorbez	::= Terminal|Bezeichner
Parameterliste	::= Parameter|Parameterliste,Parameter
Funktionsname	::= [A-Z_a-z] [^\t\n (]*
Funktion	::= [Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn] (Parameterliste)
Operator	::= [Dd][Ee][Ff][Ii][Nn][Ee] [Oo][Pp][Ee][Rr][Aa][Tt][Oo][Rr] (Operatorbez|Parameter|Klammer)+ Funktion
Operatorliste	::= Operatorbez|Operatorbez Operatorliste
Assoc	::= %left|%right|%nonassoc
Deklaration	::= Assoc Operatorliste
Zeile	::= Operator|Deklaration

3. Aufruf des Programms `op` gemäß folgender Aufrufsyntax: `op [-h] [-o newrules-file] operator-definitions-file`.

   Das Programm `op` liest die Operatordefinitionen ein und schreibt die daraus gewonnenen Grammatikregeln nach `newrules.y`.

4. Aufruf von Jacob, d.h. der Funktion `jacob()`. Bei Bedarf sind vorher die Makordefinitionen in `template.h` zu verändern. Dann ist aber Jacob erneut zu übersetzen!

   Jacob liest zuerst die Datei mit den Regeln für Standardausdrücke (`jacob.y`) und danach, wenn vorhanden, die Datei mit den neuen Grammatikregeln (`newrules.y`). Parallel dazu erstellt er die Parsetabellen.

5. Aufruf der Funktion `yyparse()`, die jetzt mit den eben erstellten Parsetabellen arbeitet.

6. bei Bedarf weiter mit 2 oder mit 5

8 Ausblick

Jacob realisiert zur Zeit durch den Aufbau des Parsebaumes nur den Syntaxcheck für einen Ausdruck mit Operatoren. Der übergeordnete Parser, der Jacob aufruft, muß also noch einen Semantikcheck vornehmen.

Außerdem wird im Moment, wenn mit einer Regel reduziert wird, der Funktionsname ausgegeben, anstatt den semantischen Wert des Symbols der linken Seite mit dieser Funktion zu berechnen. Die zu einem Operator gehörende Funktion muß, um beim Reduzieren ausgeführt werden zu können, zur Laufzeit dynamisch zum Parser dazugelinkt werden. Das ist jedoch nicht mehr Bestandteil dieser Studienarbeit, sondern soll innerhalb einer nächsten Studienarbeit realisiert werden.

Dazu einige Hinweise: Beim Reduzieren mit einer Regel wird zum Label `yyreduce` in der Funktion `yyparse()` (Datei `bison.c`) gesprungen. Dort wird zunächst der Defaultwert für das Symbol auf der linken Seite der Regel gesetzt; das ist der Wert des ersten Symbols der rechten Seite. Dann wird mit Hilfe des Vektors `yyrfunc`, der beim Einlesen der Grammatikregeln angelegt wird, ermittelt, ob zur Regel, mit der reduziert werden soll, eine Funktion existiert. Falls das der Fall ist, wird der Vektor `plist` mit den semantischen Werten der Symbole der rechten Seite gefüllt. Dieser Vektor sollte also der Funktion übergeben werden, damit die Berechnung des Wertes eines Ausdrucks basierend auf den Werten der Teilausdrücke möglich wird. Der Name der Funktion befindet sich in `func->funcname` und die Anzahl ihrer Parameter in `pno`. Das Ergebnis der Funktion wird in der Variable `yyinfo` gespeichert. Die Berechnung des semantischen Werte muß also ungefähr folgendes Aussehen haben: `yyinfo = f(plist, pno)`, statt f muß aber die jeweils gültige Funktion verwendet werden. Das Ergebnis der Funktion muß daher den Typ YYSTYPE haben, der in der Datei `template.h` definiert wird.

Die Funktion `inspect_tree()` durchläuft einmal den gesamten Baum und gibt für jeden Knoten den Wert des Infoteils aus. Dazu wird die Funktion `yyprint()` benutzt, deren Definition sich in der Datei `bison.c` befindet. Diese Funktion muß erweitert werden, wenn der Typ des Infoteils eines Knotens sich ändert.

A Unveränderte Dateien

Die folgenden Dateien wurden unverändert von Bison übernommen:

```
COPYING
alloca.c
allocate.c
closure.c
conflicts.c
derives.c
lalr.c
new.h
nullable.c
print.c
reduce.c
state.h
symtab.c
symtab.h
types.h
version.c
warshall.c
```

B Geänderte Dateien

Jede Änderung der Sourcen ist durch einen Kommentar vor der geänderten Zeile dokumentiert.[12]

Auf eine Beschreibung der Änderungen im Detail wollen wir deswegen an dieser Stelle verzichten. Wir wollen hier nur einen Überblick geben.

Die folgenden Dateien mußten gemäß Abschnitt 6 geändert werden:

- `LR0.c`

  Die extern-Deklaration für `nullable` wurde entfernt, weil diese Variable in `LR0.c` gar nicht benutzt wird.

  Eine neue Funktion, `reset_LR0()` sorgt dafür, daß alle globalen Variablen zurückgesetzt werden. (vgl. Abschnitt 6.4.3 auf Seite 24)

- `Makefile`

  Die Programme `op` und `jacob` sollen erzeugt werden, wenn `make all` ausgeführt wird. Die dafür benötigten Regeln wurden eingefügt.

  Regeln für nicht mehr existierende Dateien wurden entfernt und Regeln für neue Dateien eingefügt. Eine neue Regel sorgt dafür, daß bei Änderungen an `jacob.y` nicht `jacob.c` mit `yacc` neu erzeugt wird. (Ohne die neue Regel würde die implizite Regel `.y.c` zur Anwendung kommen.)

  Da die Funktion `yyparse()` nicht erst kopiert werden muß, bevor sie lauffähig wird (vgl. Abschnitt 6.2 auf Seite 17), wurde alles entfernt, was dafür benötigt wurde, u.a. `PFILES, PFILE` und `PFILE1`, über die dem Compiler der Name der Datei mit dem Parsegerüst mitgeteilt werden konnte.

- `bison.c`

  `bison.c` ist aus der Datei `bison.simple` hervorgegangen. Da sie nunmehr anstelle des Gerüstes für `yyparse()` eine vollständige Funktion enthält, wurde sie umbenannt.

  Das Vorgehen beim Parsen wurde übernommen, aber statt auf die statischen Parsetabellen (`yypact[ ]`, `yytname[ ]`, ...) direkt zuzugreifen, werden nun die entsprechenden Funktionen (`yypact()`, ...) verwendet (vgl. Abschnitt 6.2 auf Seite 17). `sizeof(yytname)` wurde durch `sizeof_yytname` ersetzt.

  Da `yyval` nun ein Zeiger auf einen Knoten bzw. Baum ist, kann der semantische Wert eines Symbols nicht mehr in `yyval` zwischengespeichert werden. Stattdessen wird `yyinfo` verwendet. `yylval` wird durch `leaf(yylval)` ersetzt und `yyvsp[ ]` durch `(yyvsp[..])->info` (vgl. Abschnitt 6.3.1 auf Seite 19).

  Die globale Variable `failure`, die während der Generierung der Parsetabellen auf einen Wert ungleich 0 gesetzt wird, wenn ein Fehler aufgetreten ist, wird vor Parsebeginn abgefragt, um die Funktion `yyparse()` sofort zu beenden, wenn die Tabellen

[12] So ein Kommenar beginnt immer mit `"CR:"`. Falls mehrere Zeilen hintereinander geändert wurden und nicht klar erkennbar ist, bis wo die Änderung reicht, wird dieser Bereich durch Kommentare geklammert.

nicht (oder nicht alle) angelegt wurden. Dadurch wird vermieden, daß auf nicht existierende Tabellen zugegriffen wird.

Das Makro `YYACCEPT` ruft vor Beenden des Parsens die Funktion `inspect_tree()` auf, wenn `yydebug` ungleich `0` ist.

Die Variable `func` dient zum Zwischenspeichern aller Informationen, die eine Funktion in einer Grammatikregel betreffen.

Das Dollarzeichen, an dessen Stelle beim Erstellen der Funktion `yyparse()` die Aktionen in Form einer `switch`-Anweisung kopiert wurden, wurde entfernt. Statt der Aktionen, die von Bison ausgeführt werden, wenn mit der zugehörigen Regel reduziert wurde, verwenden wir nun nur eine Zeile, die dafür sorgt, daß der Parsebaum für das Symbol auf der linken Seite der Regel aufgebaut wird.

MS-DOS spezifischer Code, `#line`-Anweisungen und Code, der nur für den wiedereintrittsfesten Parser (vgl. Abschnitt 6.5.3 auf Seite 25) ausgeführt wurde, sowie Code zwischen `#if 0` und `#endif` wurden entfernt.

Anstelle des Makros `YYLSP_NEEDED` verwenden wir die Variable `yylsp_needed`. Analog wurden `YYSTYPE` durch `YYNTYPE`, `YYFLAG` durch `MINSHORT`, `YYLAST` durch `high`, `YYFINAL` durch `final_state`, `YYNTBASE` durch `ntokens` und `YYLAST` durch `high` ersetzt.

- `files.c`

  VMS und MS-DOS- spezifischer Code wurde entfernt (vgl. Abschnitt 6.6 auf Seite 26) .

  Die Funktion `done()` wird statt mit `exit()` nun mit `longjmp()` beendet (vgl. Abschnitt 6.4.2 auf Seite 24). Die globale Variable `failure` wird gesetzt, wenn `done()` mit einem Werte ungleich Null aufgerufen wurde. Das ist notwendig, damit der Bisonparser erkennen kann, daß keine Parsetabellen angelegt wurden und ein Parsen somit nicht möglich ist. Sonst würde der Parser auf die nichtexistierenden Tabellen zugreifen und so einen Programmabsturz provozieren.

  Der Filedeskriptor `fguard` wurde nur benötigt, weil in `copy_guard()` (Datei `reader.c`) alle `guards` dorthin geschrieben wurden. Da der Gebrauch der `guards` aber nur für den semantischen Parser vorgesehen ist, wollen wir ihre Verwendung nicht mehr erlauben (vgl. Abschnitt 6.5.2 auf Seite 25) und entfernen `fguard` und `guardfile`.

  `fattrs` diente zur Aufnahme der Definitionen, die zwischen `%` und `%` stehen. Dieser C-Deklarationsteil ist künftig nicht mehr erlaubt (vgl. Abschnitt 6.2 auf Seite 17). `fattrs`, `tmpattrsfile` und `attrsfile` werden somit nicht mehr benötigt und wurden entfernt.[13]

  `definesflag` wurde entfernt, weil die Definitionsdatei nicht mehr benötigt wird. Deswegen werden auch der Filesdeskriptor `fdefines` und die Variablen `tmpdefsfile` und `defsfile`, in denen der Name einer temporären Datei bzw. der

[13] Alle Deklarationen können jetzt über die Datei `template.h` eingebunden werden.

Name der Definitionsdatei standen, entfernt. Auch das Kopieren des Inhalts der temporären Datei in die Definitionsdatei in der Funktion `done()` wurde überflüssig.

Die Funktion `open_extra_files()` wird nicht mehr benötigt, weil dort nur Dateien geöffnet wurden, die nur für den semantischen Parser benötigt werden (vgl. Abschnitt 6.5.2 auf Seite 25).

`fparser` diente als Puffer, der den Code für das Parsegerüst aus der in `filename` angegebenen Datei aufnehmen sollte, damit er dann später von dort in die Parserdatei kopiert werden konnte. `fparser` und `filename` werden also nicht mehr benötigt, weil `yyparse()` in Form einer Funktion zur Verfügung steht und nicht erst kopiert werden muß. (vgl. Abschnitt 6.2 auf Seite 17)

Nach `faction` wurden die Aktionen kopiert. Mit dem Wegfall der Aktionen werden `faction` und `actfile` überflüssig.

- `files.h`

  `PFILE` und `PFILE1`, die den Pfad zur Datei mit dem Parsegerüst (`bison.simple` oder `bison.hairy`) enthalten, werden entfernt, weil diese Pfadangaben nur zum Kopieren des `yyparse()`-Gerüstes benötigt wurden.

  `fattrs`-, `fdefines`- und `fguard`-Deklarationwn sowie `attrsfile`- und `guardfile`-Deklarationen werden ebenso wie aus `files.c` auch hier entfernt.

- `genparser.c`

  Diese Datei ist aus der Datei `main.c` hervorgegangen, die umbenannt wurde, weil dort jetzt keine `main()`-Funktion zu finden ist. Sie wurde ersetzt durch die Funktion `generate_parser()`, die jetzt immer dann aufgerufen werden kann, wenn neue Operatordefinitionen eingebunden werden sollen.

  Die Variable `program_name` wurde entfernt.

  Die Definition der Variable, die anzeigt, ob aus `finput`, dem Filedeskriptor für `jacob.y`, oder `fnewrules`, dem Filedeskriptor für `newrules.y`, zu lesen ist, wurde eingefügt.

- `getargs.c`

  `debugflag` wurde in `yydebug` umbenannt und `newrulesfile` wurde eingefügt.

  Die Funktion `getargs()` liest keinen Kommandozeilenargumente mehr ein. Wenn globale Variablen geändert werden sollen, die vorher durch Einlesen der Optionen gesetzt wurden, müssen jetzt die entsprechenden Makrodefinitionen in template.h geändert werden.

- `gram.c`

  `semantic_parser` und `pure_parser` wurden entfernt (vgl. Abschnitt 6.5.2 auf Seite 25).

  VMS-spezifischer Code wurde entfernt (vgl. Abschnitt 6.6 auf Seite 26).

- `gram.h`

  Die extern-Deklarationen für `semantic_parser` und `pure_parser` (vgl. Abschnitt 6.5.2 auf Seite 25) wurden entfernt.

- `lex.c`

  Die bedeutendste Änderung in `lex.c` ist die Ersetzung von `getc()` durch das Makro `GETC` bzw. von `ungetc()` durch das Makro `UNGETC` (vgl. Abschnitt 6.1 auf Seite 16).

  `%no_lines`, das nur beim Kopieren des Parsegerüstes benötigt wurde, wenn keine `#line`-Precompileranweisungen geschrieben werden sollten, ist nicht mehr zulässig und bekommt deshalb in der Struktur `percent_table` nicht mehr den Zeiger auf `nolinesflag`, sondern einen NULL-Zeiger, d.h. `%no_lines` wird jetzt ignoriert, wenn es in einer Grammatikdatei gefunden wird (vgl. Abschnitt 6.5.2 auf Seite 25 und Abschnitt 6.5.3 auf Seite 25).

  Die Funktion `lex()` gibt jetzt den Tokentyp `FUNCTION` zurück, wenn das Schlüsselwort `function` in der Grammatikdatei gefunden wurde.

  Ein Block, der in `#if 0` und `#endif` geschachtelt war, wurde der besseren Übersicht halber entfernt. Da `definesflag`, `noparserflag`, `fixed_outfiles`, `verboseflag`, `debugflag`, `spec_name_prefix` und `spec_file_prefix` nur innerhalb dieses Blockes verwendet wurden, sind die `extern`-Deklarationen dafür überflüssig und konnten entfernt werden.

- `lex.h`

  Einen neue `define`-Anweisung für `FUNCTION` wurde eingefügt.

- `machine.h`

  MS-DOS-spezifischer Code wurde entfernt und zwei Kommentare wurden eingefügt.

- `output.c`

  `yystos[ ]` mit der Funktion `output_stos()` wurden nur für den semantischen Parser benötigt und deswegen entfernt [14] (vgl. Abschnitt 6.5.2 auf Seite 25). `GUARDSTR` und `attrsfile` werden aus dem selben Grund nicht mehr nach `fguard` geschrieben.

  Alle Makros, die zum Gebrauch im `yyparse()`-Gerüst oder in den Aktionen bestimmt waren und in die Parserdatei geschrieben wurden, sind nun überflüssig. Anstelle der Makros müssen nun die Variablen verwendet werden, deren Werte die Makros repräsentierten. Tabelle 3 zeigt die betroffenen Variablen.

  Die Funktion `output_defines()` hatte als einzige Aufgabe, Definitionen für Makros in das Tabfile zu schreiben. Sie ist also überflüssig und demzufolge entfernt worden.

  Die Funktionen `output_headers()` und `output_trailers()`, die benutzt wurden, um die `switch`-Anweisung für die Aktionsrümpfe zu erstellen und um Definitionen

[14] Deswegen wurde auch der Kommentar am Dateianfang geändert, der die verwendeten Tabellen erklärt. Mit einem Stern waren die Tabellen gekennzeichnet, die nur für den semantischen Parser erzeugt wurden und mit zwei Sternen die, die nur erzeugt wurden, wenn bestimmte Schalter gesetzt sind. Da die Tabelle für den semantischen Parser (`yystos[ ]`) nun grundsätzlich nicht mehr erzeugt wird, gibt es jetzt nur noch einen einfachen Stern zur Kennzeichnung der schalterabhängigen Tabellen.

Makro	Variable
YYDEBUG	yydebug
YYFINAL	final_state
YYFLAG	Makro MINSHORT
YYNTBASE	ntokens
YYNTOKENS	ntokens
YYNNTS	nvars
YYNRULES	nrules
YYNSTATES	nstates
YYMAXUTOK	max_user_token_number
YYLAST	high

Tabelle 3: Verwendung von Variablen anstelle von Makros

für yyparse, yylex, yyerror, yylval, yychar, yydebug, yynerrs zu schreiben, sowie die Guards zu kopieren, wurden entfernt.

Für die dynamische Verwaltung der Parsetabellen (vgl. Anschnitt 6.2 auf Seite 17) sind einige neue Funktionen und Variablen nötig. Das sind pro Tabelle:

- ein Vektor, in den die Tabelleneinträge geschrieben werden:

  yytranslate_table, yyprhs_table, yyrhs_table, yyrline_table, yytname_table, yytoknum_table, yyr1_table, yyr2_table, yydefact_table, yydefgoto_table, yypact_table, yypgoto_table, yytable_table, yycheck_table

- eine Variable für die aktuelle Tabellengröße:

  csize_translate, csize_prhs, csize_rhs, csize_rline, csize_tname, csize_toknum, csize_r1, csize_r2, csize_defact, csize_defgoto, csize_pact, csize_pgoto, csize_table, csize_check

- eine Variable für die maximale Tabellengröße:

  maxsize_translate, maxsize_prhs, maxsize_rhs, maxsize_rline, maxsize_tname, maxsize_toknum, maxsize_r1, maxsize_r2, maxsize_defact, maxsize_defgoto, maxsize_pact, maxsize_pgoto, maxsize_table, maxsize_check

- Anstatt wie vorher auf die Tabellen über Vektoren zuzugreifen, gibt es jetzt neue Funktionen, die die Werte aus den neuen Tabellen auslesen:

  yytranslate(), yyprhs(), yyrhs(), yyrline(), yytname(), sizeof_yytname(), yytoknum(), yyr1(), yyr2(), yydefact(), yydefgoto(), yypact(), yypgoto(), yytable(), yycheck().

  Dadurch, daß wir Funktionen verwenden, anstatt auf die Arrays direkt zuzugreifen, können die Tabellen in output.c statisch bleiben und sind vor ungewollten Veränderungen geschützt, und die Aufrufsyntax bleibt ähnlich – ein Auslesen von yytranslate[ ] beispielsweise wird jetzt durch einen Aufruf von yytranslate() ersetzt.

Folgende Funktionen, die die Tabellen vorher in die Parserdatei geschrieben haben, wurden durch die dynamische Variante ersetzt: `output_token_translations()`, `output_gram()`, `output_rule_data()`, `token_actions()`, `goto_actions()`, `output_base()`, `output_table()` und `output_check()`.

Da in der Grammatikdatei kein C-Code mehr erlaubt ist (vgl. Abschnitt 6.2 auf Seite 17), konnten die Funktion `output_program()`, die den C-Code nach dem zweiten `%%` kopiert hat und die Funktion `output_parser()`, die das Parsegerüst kopiert hat, entfernt werden. Die C-Deklarationen zwischen `%{` und `%}` wurden entweder nach `fattrs` kopiert und von dort in das Tabfile geschrieben oder, wenn ein semantischer Parser erzeugt werden sollte, in eine Extradatei, `attrsfile`, geschrieben, die dann über eine Includeanweisung in das Tabfile eingebunden wurde. Beide Varianten wurden entfernt.

Die Deklaration für die Funktion `free_itemset()` wurde entfernt, weil diese Funktion gar nicht existierte (hingegen gibt es eine Funktion `free_itemsets()`).

Die Funktion `reader_output_yylsp()` wurde entfernt, weil sie nur zum Schreiben von `LTYPESTR`, der das Makro `YYLTYPE` definierte, in das Tabfile benutzt wurde. Diese Definition steht jetzt in `jacob.h`.

Eine Includeanweisung für `stdio.h` und eine Anweisung, die dafür sorgt, daß `const` nichts tut, wenn der Compiler nicht ANSI-C verwendet, werden jetzt nicht mehr in die Parserdatei geschrieben und stehen stattdessen in `jacob.h`.

Ein Umbenennen von Variablen und Funktionen, das bei Bison über die Kommandozeilenoption -p möglich ist, funktioniert bei Jacob nur mit Hilfe von Makrodefinitionen in `template.h`, was eine Neuübersetzung des Programmcodes erfordert. Dafür wurde alles aus der Funktion `output_headers()` entfernt, was dazu diente, Makros für `yyparse()`, `yylex()`, `yyerror()`, `yylval`, `yychar`, `yydebug` und `yynerrs` in das Tabfile zu schreiben. Stattdessen wurde `template.h` eingebunden, weil die Variable `yydebug` hier verwendet wird.

- `reader.c`

  Aktionen in der Grammatikdatei sind nicht mehr erlaubt (vgl. Abschnitt 6.2 auf Seite 17). Deswegen wurde die Funktion `copy_action()` so verändert, daß alles zwischen { und } ignoriert und eine Warnung herausgegeben wird, wenn eine Aktion gefunden wurde. Das Kopieren des Aktionsrumpfes nach `faction` unterbleibt. `actionflag` und `xactions` sowie Code, der Aktionen, die in der Regelmitte standen, behandelte, wurden durch `functionflag` ersetzt.

  Statt der Aktionen kann der Nutzer nun Funktionen definieren, die ausgeführt werden, wenn mit der zugehörigen Regel reduziert wird. Dafür wurden die Variablen `rfunc` (Vektor, der durch die Regelnummer indiziert wird und die Funktionsnamen enthält) und `rfunc_allocated` (Größe des für `rfunc` allozierten Speichers) eingeführt. Die Funktion `parse_function()` erledigt das Einlesen einer Funktionsdefinition und `record_function()` füllt den Vektor `rfunc`.

  In Übereinstimmung mit Abschnitt 6.5.2 und Abschnitt 6.5.3 auf Seite 25 wurde alles entfernt, was nur für den semantischen oder den wiedereintrittsfesten Parser

benötigt wurde: die Variablen `semantic_parser` und `pure_parser`, die Funktionen `open_extra_files()` und `copy_ guard()`, in `read_declarations()` die Zweige `SEMANTIC_PARSER` und `PURE_PARSER` [15]

Blöcke zwischen `#if 0` und `#endif` sowie drei Fälle, die nie eintreten konnten, wurden ebenso wie die `extern`-Deklaration für `done()` entfernt.

Alles, was der C-Codegenerierung als Zwischendarstellung diente, wurde entfernt. Dazu gehören das Schreiben des Anfangskommentars, der Definition für `YYBISON`, der Defaultdefinition für `YYSTYPE`, der C-Deklarationen und der Tokendefinitionen in die Parserdatei bzw. die Definitionsdatei. Somit konnten auch `definesflag`, das anzeigte, ob die Definitionsdatei erzeugt werden sollte, sowie die Funktionen `copy_definition()`, die die C-Deklarationen zwischen `%{` und `%}` in das Tabfile kopierte, und `output_token_defines()`, die die Tokendefinitionen in die Definitionsdatei kopierte, entfernt werden. Statt die Funktion `copy_definition()` aufzurufen, wird jetzt eine Fehlermeldung erzeugt, wenn C-Deklarationen in der Grammatikdatei gefunden werden.

Da auch die `%union`-Deklaration nicht mehr zulässig ist, wurde die Funktion `parse_union_decl()` entfernt.

Die Funktion `reader_output_yylsp()` diente nur zum Schreiben des Makros `LTYPESTR`, das `YYLTYPE` definierte, in das Tabfile. Die Funktion und somit auch das Makro konnten entfernt werden. Die Definition für `YYLTYPE` steht jetzt in `jacob.h`.

Nach dem Lesen der Eingabedatei wird jetzt versucht, aus `newrules.y` zu lesen, wenn vorhanden. Dazu werden `t`, `lineno` und `infile` gesichert, um diese nach Lesen aus `newrules.y` wieder zurücksetzen zu können. Statt `getc()` muß jetzt `GETC()` `verwendet` werden und statt `ungetc()` `UNGETC()` (vgl. Abschnitt 6.1 auf Seite 16).

Die Warnung für eine zweites `%prec` innerhalb einer Regel wurde nie erreicht. Deshalb benutzen wir `precflag`, um die Warnung gegebenenfalls an anderer Stelle auszugeben.

- `system.h`

  MS-DOS- und VMS-spezifischer Code wurde entfernt (vgl. Abschnitt 6.6 auf Seite 26).

[15] `%semantic_parser` und `%pure_parser` führen jetzt zu einer Fehlermeldung.

C Überflüssige Dateien

Folgende Dateien werden für Jacob nicht mehr benötigt:

- die Dateien mit Dokumentationen `ChangeLog`, `INSTALL`, `Makefile.in`, `NEWS`, `README`, `REFERENCES` und `bison.1`
- die VMS-Dateien `bison.cld`, `bison.rnh`, `build.com`, `vmsgetargs.c` und `vmshlp.mar`
- das Parsegerüst des semantischen Parsers `bison.hairy`
- die Infoseiten zu Bison `bison.info`, `bison.info-1`, `bison.info-2`, `bison.info-3`, `bison.info-4` und `bison.info-5` sowie deren Sourcecode `bison.texinfo`
- die Dateien mit dem Parsegerüst des einfachen Parsers `bison.simple` und `bison.s1`
- die Installationsdateien `config.cache`, `config.log`, `config.status`, `configure`, `configure.bat`, `configure.in`, `install-sh` und `mkinstalldirs`
- die Dateien `getopt.c`, `getopt1.c` und `getopt.h`, die die Funktion `getopt()` für GNU zur Verfügung stellen und nicht mehr gebraucht werden, weil `getopt()` jetzt Teil der C-Bibliothek ist
- `texinfo.tex`, das Texfile mit Makros, um mit texinfo-Dateien umzugehen

D Neue Dateien

Folgende Dateien wurden neu erstellt:

- `jacob.c`

  Diese Datei enthält im wesentlichen eine `main()`-Funktion für das Programm `jacob` sowie die Parsergenerierungsfunktion `jacob()`.

- `jacob.h`

  Eine Includeanweisung für `stdio.h` und eine Anweisung, die dafür sorgt, daß `const` nichts tut, wenn der Compiler nicht ANSI-C verwendet, werden jetzt nicht mehr in die Parserdatei geschrieben und stehen stattdessen hier.

- `jacob.l`

  Diese Datei definiert den Aufbau der syntaktischen Elemente eines Ausdrucks. Sie dient als Eingabe für `flex`.

  Außerdem wird hier die Fehlerreportfunktion `yyerror()` für Jacob definiert. Da diese Funktion die aktuelle Eingabezeile sowie einen Zeiger (`^`) auf den Anfang des aktuellen Tokens ausgibt, wird bei jedem Zeilenumbruch die Zeile in den Puffer `linebuf` geschrieben und bei jedem gelesenen Token in der Variable `tokenpos_save` der Beginn des Tokens gespeichert.

- `jacob.y`

  In dieser Datei befinden sich die Regeln für Standardausdrücke sowie Tokendefinitionen. Sie wird von der Funktion vor der Datei `newrules.y` eingelesen.

- `modgetc.h`

  Hier werden die Makros `GETC` und `UNGETC` definiert, die dafür sorgen, daß aus `fnewrules`, d.h. aus der durch das Programm `op` erzeugten Datei, anstatt aus `finput` gelesen wird, wenn `use_modified_getc` gesetzt ist.

- `newrules.y`

  Diese Datei wird vom Programm `op` aus der Datei `operatoren.def` erzeugt und enthält Grammatikregeln für Ausdrücke, die nutzerdefinierte Operatoren enthalten sowie eventuell Angaben zum Vorrang der Operatoren (`%left`, `%right` und `%nonassoc`, die im Gegensatz zu Bisondeklarationen mit `%` abgeschlossen sind).

- `op.h`

  Diese Datei enthält die Definition für `YYSTYPE`, den Typ von `yylval`.

- `op.l`

  Diese Date i dient als Eingabe für `flex` und definiert den Aufbau der syntaktischen Elemente einer Operatordefinition. Außerdem wird hier die Fehlerreportfunktion `yyerror()` für das Programm `op` definiert. Alles was über diese Funktion in der Beschreibung der Datei `jacob.l` steht, trifft auch hier zu.

- `op.y`

  Diese Datei dient als Eingabe für Bison (den nicht modifizierten, weil wir die Aktionen benötigen). Hier finden sich die Grammatikregeln, die den Aufbau einer Operatordefinition beschreiben sowie die Funktionen `done()`, `tryopen()` (die beide aus den gleichnamige Bison-Funktionen hervorgegangen sind) und eine `main()`-Funktion, die Kommandozeilenoptionen auswertet, Dateideskriptoren öffnet und die Parsefunktion aufruft.

- `operatoren.def`

  In dieser Datei finden sich Beispiele für Operatordefinitionen. Diese müssen bei Bedarf entfernt und durch neue ersetzt werden.

- `output.h`

  In dieser Datei befinden sich `extern`-Deklarationen für die Funktionen, die die Werte der dynamisch erzeugten Parsetabellen liefern (vgl. die Beschreibung der Änderungen in der Datei output.c auf Seite 34).

- `template.h`

  Die Tokendefinitionen befinden sich hier.

  Eine Typdefinition für `funcdesc` wurde eingefügt. Der Typ `funcdesc` dient zum Beschreiben einer Funktion innerhalb einer Grammatikregel.

  Der Typ von `yylval` (`YYSTYPE`) wird nun hier festgelegt. Ebenso können die C-Deklarationen, die in einer Bisongrammatikdatei zwischen `%{` und `%}` standen, nun über `template.h` eingebunden werden.

  `YYERROR_VERBOSE` wird hier definiert.

  Wenn ein anderes Präfix als `yy` gewünscht wird, sind für `yyparse`, `yylex`, `yyerror`, `yylval`, `yychar`, `yydebug`, `yynerrs` Makrodefinitionen in die Datei `template.h` zu schreiben, z.B. `#define yyparse xxparse`.

- `tree.c`

  In dieser Datei befinden sich drei Funktionen, die das Konzept des Parsebaumes umsetzen.

  Die Funktion `leaf()` liefert einen Zeiger auf einen Baum, der aus einem einzigen Knoten besteht, d.h. auf ein Blatt, und erwartet einen Parameter vom Typ `info_t`, d.h. den eigentlichen semantischen Wert eines Symbols. Sie wird beim Schieben eines Tokens benötigt.

  Die Funktion `tree()` formt aus mehreren Teilbäumen und dem Infoteil des Wurzelknotens einen neuen Baum. Sie liefert auch einen Zeiger auf einen Knoten, der in diesem Fall die Wurzel des erzeugten Baumes darstellt, und erwartet als Parameter eine Liste von Zeigern auf die Wurzelknoten der Teilbäume, die Anzahl der Teilbäume und den Infoteil des Wurzelknotens. Das sind genau die Informationen, die beim Reduzieren zur Verfügung stehen, wenn der Baum für das Nichtterminal der linken Seite einer Regel erzeugt werden soll.

Die Funktion `inspect_tree()` durchläuft einmal den gesamten Baum und gibt für jeden Knoten den Wert des Infoteils aus.

- `tree.h`

  Hier befinden sich die `extern`-Deklarationen der in `tree.c` definierten Funktionen.

E Literatur

[ASU88] AHO, Alfred V.; SETHI, Ravi; ULLMANN, Jeffrey D.: *Compilerbau.* Addison-Wesley, 1988.

[Sha97] SHANG, David L.: *Transframe: A White Paper.* http://www.transframe.com/transframe/tf_white.htm, 1996-1997

[KV84] KLINT, Paul; VISSER, Eelco: *Using Filters for the Disambiguation of Context-free Grammars.* University of Amsterdam, 1984.